Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Computation 199 (2005) 200–227
Information and Computation
www.elsevier.com/locate/ic

Proving pointer programs in higher-order logic

Farhad Mehta[a,*,1], Tobias Nipkow[b]

[a] *Department of Computer Science, ETH Zürich, Switzerland*
[b] *Institut für Informatik, Technische Universität München, Germany*

Received 17 November 2003; revised 29 July 2004
Available online 28 January 2005

Abstract

Building on the work of Burstall, this paper develops sound modelling and reasoning methods for imperative programs with pointers: heaps are modelled as mappings from addresses to values, and pointer structures are mapped to higher-level data types for verification. The programming language is embedded in higher-order logic. Its Hoare logic is derived. The whole development is purely definitional and thus sound. Apart from some smaller examples, the viability of this approach is demonstrated with a non-trivial case study. We show the correctness of the Schorr–Waite graph marking algorithm and present part of its readable proof in Isabelle/HOL.

Keywords: Pointer programs; Verification; Hoare logic; Higher-order logic; Schorr–Waite algorithm

1. Introduction

It is a truth universally acknowledged, that the verification of pointer programs must be in want of machine support. The basic idea in all approaches to pointer program proofs is the same and

* Corresponding author. Fax: +41 44 632 1435.
E-mail address: fmehta@inf.ethz.ch (F. Mehta).
[1] Work done while at the Technische Universität München, supported by DFG Grant NI 491/6-1.

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2004.10.007

goes back to Burstall [6]: model the heap as a collection of variables of type $address \to value$ and reason about the programs in Hoare logic. A number of refinements of this idea have been proposed; see [16] for a partial bibliography. The most radical idea, again inspired by Burstall, is that of *separation logic* [17]. Although very promising, it is difficult to combine with existing theorem proving infrastructure because of its special logical connectives. Instead we take Bornat's [3] presentation of Burstall's ideas as our point of departure.

Systematic approaches to automatic or interactive verification of pointer programs come in two flavours. There is a large body of work on program analysis techniques for pointer programs. These are mainly designed for use in compilers and can only deal with special properties like aliasing. In the long run these approaches will play an important role in the verification of pointer programs. But we ignore them for now because our goal is a general purpose logic. For the same reason we do not discuss other special purpose logics, e.g. [9].

General theorem proving approaches to pointer programs are few. A landmark is the thesis by Suzuki [20] who developed an automatic verifier for pointer programs that could handle the Schorr–Waite algorithm. However, that verification is based on 5 recursively defined predicates (which are not shown to be consistent – mind the recursive "definition" $P = \neg P$!) and 50 unproved lemmas about those predicates. Bornat [3] has verified a number of pointer programs with the help of Jape [4]. However, his logical foundations are a bit shaky because he choses not to address definedness and soundness issues, leaving many lemmas unproven. Furthermore, since Jape is only a proof editor with little automation, the Schorr–Waite proof takes 152 pages [5]. Apart from the works of Suzuki and Bornat, we are unaware of any other mechanically checked proofs of the Schorr–Waite algorithm.

The contributions of our paper are as follows:

- An embedding of a Hoare logic for pointer programs in a general purpose theorem prover (Isabelle/HOL).
- A logically fully sound method for the verification of inductively defined data types like lists and trees on the pointer level.
- A treatment of cyclic pointer structures using similar methods.
- A readable and machine checked proof of the Schorr–Waite algorithm.

Instrumental in achieving the second point is the replacement of Bornat's recursive abstraction functions, which require a logic of partial functions, by inductive relations, which are definable in a logic of total functions. This is crucial, as most exiting theorem provers support total functions only.

The point about "readable" proofs deserves special discussion as it is likely to be controversial. Our aim was to produce a proof that is close to a journal-style informal proof, but written in a stylised proof language that can be machine-checked. Isabelle/Isar [21,11], like Mizar, provides such a language. Publishing this proof should be viewed as creating a reference point for further work in this area: although an informal proof is currently shorter and more readable, our aim should be to bridge this gap further. It also serves as a reference point for future mechanisations of other formal proofs like the separation logic one by Yang [22].

The rest of the paper is structured as follows. After a short overview of Isabelle/HOL (Section 2) and an embedding of a simple imperative programming language in Isabelle/HOL (Section 3), we

describe how we have extended this programming language with references (Section 4). We show in some detail how to prove programs involving linked lists (Section 5) and present a number of verification examples (Section 6) illustrating our approach. We then present some extensions (Section 7), including how our approach works for trees (Section 7.3). Finally we present our main case study, the structured proof of the Schorr–Waite algorithm (Section 8).

2. Isabelle/HOL

Isabelle/HOL [12] is an interactive theorem prover for HOL, higher-order logic. The whole paper is generated directly from the Isabelle input files, which include the text as comments. That is, if you see a lemma or theorem, you can be sure its proof has been checked by Isabelle.

2.1. Isabelle/HOL notation

Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of the nons-tandard notation.

The space of total functions is denoted by the infix $\Rightarrow$. Other type constructors, e.g., *set*, are written postfix, i.e., follow their argument as in $'a\ set$ where $'a$ is a type variable.

The syntax $[\![P;\ Q]\!] \Longrightarrow R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow R$. Note that semicolon will also denote sequential composition of programs, which should cause no major confusion. There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing, except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. Beware that $\longrightarrow$ binds more tightly than $\Longrightarrow$: in $\forall x.\ P \longrightarrow Q$ the $\forall x$ covers $P \longrightarrow Q$, whereas in $\forall x.\ P \Longrightarrow Q$ it covers only P.

A HOL speciality is its ε-operator: $SOME\ x.\ P\ x$ is an arbitrary but fixed x that satisfies P. If there is no such x, an arbitrary value is returned — note that all HOL types are non-empty! HOL provides the notation $f(a := v)$ for updating function f at argument a with the new value v. Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$ and is also available for tuples, e.g. $\{(x, y, z).\ P\}$. Lists in HOL are of type $'a\ list$ and are built up from the empty list [] via the infix constructor # for adding an element at the front. In the case of non-empty lists, functions *hd* and *tl* return the first element and the rest of the list, respectively. Two lists are appended with the infix operator @. Function *set* turns a list into a set, function *rev* reverses a list. Function *distinct* of a list is true iff all its elements are pairwise distinct.

2.2. Automation in Isabelle/HOL

The automatic proofs in this paper rely almost exclusively on simplification, predicate calculus reasoning, and a combination of the two. Simplification is an extended version of higher-order conditional term rewriting. Its features are described in [12]. There are two tableau-style provers [14,15] both of which are generic in the sense that they can be extended with new inference rules and are not restricted to predicate calculus. In addition they know about sets and relations and are

extended with further concepts as we go along. One of the provers is combined with simplification by interleaving it with search in an ad-hoc way. Although the technique is incomplete, it turns out to be quite successful in practice. When describing a proof as "automatic" this is short for saying that it follows by a single invocation of the combined tableau prover and simplifier.

To make lemmas available to the automatic proof methods they can be annotated with the following attributes which we show to the right of the lemma:

- [*simp*] means it is automatically used as a simplification rule.
- [*iff*] means it is used both as a simplification rule and as a logic equivalence by the tableau provers.
- [*intro*] means it is used as an introduction rule by the tableau provers, i.e., for back-chaining.
- [*intro*!] is the same as [*intro*] except that search does not backtrack over an application of this rule.

Lemmas can also be added locally to each invocation of a proof method.

3. A simple programming language

In the style of Gordon [8] we define a little programming language and its operational semantics. The basic constructs of the language are assignment, sequential composition, *if-then-else* and *while*. The rules of Hoare logic (for partial correctness) are derived as theorems about the semantics and are phrased in a weakest precondition style. To automate their application, a proof method *vcg* has been defined in ML. It turns a Hoare triple into an equivalent set of HOL formulae (i.e., its verification conditions). This requires that all loops in the program are annotated with invariants. More semantic details can be found elsewhere [13]. Here is an example:

```
multiply-by-add:
 VARS m s a b::nat
 {a=A ∧ b=B}
 m := 0; s := 0;
 WHILE m ≠ a
 INV {s=m*b ∧ a=A ∧ b=B}
 DO
  s := s+b; m := m+1
 OD
 {s = A*B}
```

The program performs multiplication by successive addition. The first line declares the program variables m, s, a, b, (of type *nat*) to distinguish them from the auxiliary variables A and B. In the precondition A and B are equated with a and b – this enables us to refer to the initial value of a and b in the postcondition.

The statement *multiply-by-add* is a lemma to be proven. The application of *vcg* leaves three subgoals: the validity of the invariant after initialisation of m and s, preservation of the invariant, and validity of the postcondition upon loop termination. All three are proved automatically using simplification and linear arithmetic.

1. $\bigwedge m\ s\ a\ b.\ a = A \wedge b = B \Longrightarrow 0 = 0 * b \wedge a = A \wedge b = B$
2. $\bigwedge m\ s\ a\ b.$
   $(s = m * b \wedge a = A \wedge b = B) \wedge m \neq a \Longrightarrow$
   $s + b = (m + 1) * b \wedge a = A \wedge b = B$
3. $\bigwedge m\ s\ a\ b.\ (s = m * b \wedge a = A \wedge b = B) \wedge \neg\ m \neq a \Longrightarrow s = A * B$

4. References and the heap

This section describes how we model references and the heap. We distinguish *addresses* from *references*: a reference is either null or an address. We will use the term location with address, and the term pointer with reference interchangeably. Formally we declare a new unspecified type *addr* of addresses and define:

datatype *ref* = *Null* | *Ref addr*

A simpler model is to declare a type of references with a constant *Null*, thus avoiding *Ref* and *addr*. We found that this leads to slightly shorter formulae but slightly less automatic proofs, i.e., it makes very little difference.

The following properties are easy to prove by case distinction, and are made available to Isabelle's automated proof tactics.

$(x \neq Null) = (\exists y.\ x = Ref\ y)$ [*iff*]
$(\forall y.\ x \neq Ref\ y) = (x = Null)$ [*iff*]

Function *addr* :: *ref* ⇒ *addr* unpacks *Ref*, i.e. :

$addr\ (Ref\ a) = a$ [*simp*]

Our model of the heap follows Bornat [3] and is suitable for encoding fields of records or classes as in C, Pascal, or Java. We have one heap f of type *address* → *value* for each field name f. Using function update notation, an assignment of value v to field f of a record pointed to by reference r is written $f := f((addr\ r) := v)$, and access of f is written $f(addr\ r)$. Based on the syntax of Pascal, we introduce some more convenient notation:

$f(r \rightarrow e) = f((addr\ r) := e)$
$r\hat{}.f := e = f := f(r \rightarrow e)$
$r\hat{}.f = f(addr\ r)$

Note that the rules are ordered: the last one only applies if the previous one does not apply, i.e., if it is a field access and not an assignment.

To give a taste of the syntax for assertions and commands just defined, we end with a trivial example involving pointer updates and multiple de-referencing, due to Suzuki [20]. Note that *val* and *next* are record fields (i.e., heaps), and w, x, y, z are ordinary program variables.

VARS val next w x y z
{$w = Ref\ a \wedge x = Ref\ b \wedge y = Ref\ c \wedge z = Ref\ d \wedge distinct[a,b,c,d]$}
$w\hat{}.next := x;\ x\hat{}.next := y;\ y\hat{}.next := z;\ x\hat{}.next := z;$

$w\hat{}.val := (1::int);\ x\hat{}.val := 2;\ y\hat{}.val := 3;\ z\hat{}.val := 4$
$\{\ w\hat{}.next\hat{}.next\hat{}.val = 4\ \}$

The verification condition generated is uninteresting given that it merely involves function updates, a trivial task for the simplifier using the rule:

$$(f(x := y))\ z = (if\ z = x\ then\ y\ else\ f\ z) \qquad [simp]$$

Since a, b, c, and d are distinct the proof is automatic.

The above example deals with a linked list of length four. The assertions are therefore quite straightforward. More subtle reasoning is needed to express linked lists of arbitrary length. The next section is devoted to this.

5. Lists on the heap

The general approach to verifying low level structures is *abstraction*, i.e., mapping them to higher level concepts. Linked lists are represented by their 'next' field, i.e., a heap of type:

types $next\ =\ addr \Rightarrow ref$

An abstraction of a linked list of type *next* is a HOL list of type *addr list*.

5.1. Naïve functional abstraction

The obvious abstraction function *list* has type $next \Rightarrow ref \Rightarrow addr\ list$, where the second parameter is the start reference, and is defined as follows:

$$list\ next\ Null = []$$
$$list\ next\ (Ref\ a) = a\ \#\ list\ next\ (next\ a)$$

However, this is not a legal definition in HOL because HOL is a logic of total functions but function *list* is not total: *next* could contain a loop or an infinite chain. We will now examine an alternative definition.

5.2. Relational abstraction

Instead of functions we work with relations: *List next x as* means that *as* is a list of addresses that connects x to *Null* by means of the *next* field. The predicate *List* is therefore a relational abstraction for acyclic lists and can be defined using primitive recursion on the list of addresses:

$$List :: next \Rightarrow ref \Rightarrow addr\ list \Rightarrow bool$$

$$List\ next\ r\ [] = (r = Null) \qquad [simp]$$
$$List\ next\ r\ (a\#as) = (r = Ref\ a \wedge List\ next\ (next\ a)\ as) \qquad [simp]$$

Additionally, the following properties are proven using case distinction on *as* and are made available to Isabelle's automated proof tactics:

$$List\ next\ Null\ as = (as = [\,]) \quad [simp]$$
$$List\ next\ (Ref\ a)\ as = (\exists\, bs.\ as = a \# bs \wedge List\ next\ (next\ a)\ bs) \quad [simp]$$

We will now discuss the basic properties of this relation. By induction on *as* we can show:

$$a \notin set\ as \Longrightarrow List\ (next(a := y))\ x\ as = List\ next\ x\ as \quad [iff]$$

which, in the spirit of Bornat [3], is an important *separation lemma*: it says that updating an address that is not part of some linked list does not change the list abstraction. This allows us to localise the effect of assignments.

An induction on *as* shows that *List* is in fact a function:

$$[\![List\ next\ x\ as;\ List\ next\ x\ bs]\!] \Longrightarrow as = bs$$

and that any suffix of a list is a list:

$$List\ next\ x\ (as\ @\ bs) \Longrightarrow \exists\, y.\ List\ next\ y\ bs$$

Thus a linked list starting at *next a* cannot contain *a*:

$$List\ next\ (next\ a)\ as \Longrightarrow a \notin set\ as \quad [simp]$$

otherwise *as* could be decomposed into $bs\ @\ a \# cs$ and then the previous two lemmas lead to a contradiction. It follows by induction on *as* that all elements of a linked list are distinct:

$$List\ next\ x\ as \Longrightarrow distinct\ as \quad [simp]$$

This is not the end of our heap abstraction relations. In the following section we introduce the notions of a path, and a non-repeating path in the heap, and in Section 7.1, we revisit functional abstraction for lists.

6. Verification examples

After this collection of essential lemmas, we will now turn to real program proofs in this section. Motivated by examples, new predicates on the heap will be introduced along with their essential properties.

6.1. In-place list reversal

We start with the acyclic in-place list reversal algorithm:

linear-list-rev:
VARS next p q tmp
{*List next p Ps*}
$q := Null$;
WHILE $p \neq Null$
INV {$\exists\, ps\ qs.\ List\ next\ p\ ps \wedge List\ next\ q\ qs \wedge$

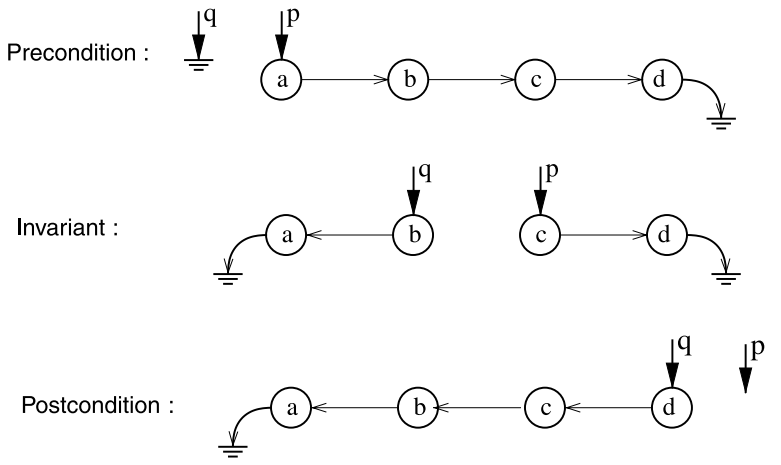

Fig. 1. In-place list reversal.

$$\begin{array}{l} \quad set\ ps \cap set\ qs = \{\} \wedge rev\ ps\ @\ qs = rev\ Ps\} \\ DO\ tmp := p;\ p := p\hat{}.next;\ tmp\hat{}.next := q;\ q := tmp\ OD \\ \{List\ next\ q\ (rev\ Ps)\} \end{array}$$

Fig. 1 illustrates the above algorithm. The precondition states that *Ps* is a list in the heap pointed to by *p*. The accumulator list is pointed to by *q*, which is initially set to *Null*. At the end, the list starting at *q* is *rev Ps*: *Ps* has been reversed onto the accumulator list starting at *q*. The invariant is existentially quantified because we have no way of naming the intermediate lists. Program variable *tmp* serves as temporary storage for a pointer swap.

The proof of linear list reversal is trivial: the Hoare triple is transformed into three HOL verification conditions using *vcg*, each of which is automatically proven.

6.2. In-place list merge

We now come to the problem of merging two sorted lists. Analogous to *rev*, our specification of correctness will be the function *merge* in Isabelle, which merges two ordered lists into a single ordered list.

$$merge :: 'a\ list * 'a\ list * ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list$$

The last argument to merge is a suitable partial ordering function. For this example each list element needs to be associated with a value. Similar to *next*, we need a pointer field *val*, that maps addresses to some type with the partial order $\leqslant$ defined over it. Since we work with lists of type *addr*, the partial ordering $\leqslant$ needs to be lifted to this type:

$$ord\ val\ x\ y \equiv val\ x \leqslant val\ y$$

The in-place list merge algorithm, without the statement of the invariant is:

list-merge:
VARS val next p q r s
{*List next p Ps* ∧ *List next q Qs* ∧ *set Ps* ∩ *set Qs* = {} ∧ (*p* ≠ *Null* ∨ *q* ≠ *Null*)}
IF q = *Null* ∨ (*p* ≠ *Null* ∧ *p*ˆ.*val* ⩽ *q*ˆ.*val*)
THEN r := *p*; *p* := *p*ˆ.*next*
ELSE r := *q*; *q* := *q*ˆ.*next FI*;
s := *r*;
WHILE p ≠ *Null* ∨ *q* ≠ *Null*
INV { *Merge-Inv* }
DO IF q = *Null* ∨ (*p* ≠ *Null* ∧ *p*ˆ.*val* ≤ *q*ˆ.*val*)
*THEN s*ˆ.*next* := *p*; *p* := *p*ˆ.*next*
*ELSE s*ˆ.*next* := *q*; *q* := *q*ˆ.*next FI*;
s := *s*ˆ.*next*
OD
{*List next r* (*merge*(*Ps*,*Qs*,*ord val*))}

Fig. 2 illustrates this algorithm. The precondition states that *Ps* and *Qs* are two disjoint lists starting at *p* and *q*. During runtime, pointers *p* and *q* move down their respective lists, leaving all visited locations in sorted order as a list segment from pointer *r* to pointer *s*. Finally, *p* and *q* are *Null*, and *r* points to the resulting merged list.

Since our loop invariant needs to talk about list segments in addition to lists, it turns out to be convenient to define an additional, more general relation: *Path next x as y* means that *as* is a path of addresses that connects *x* to *y* by means of the *next* field.

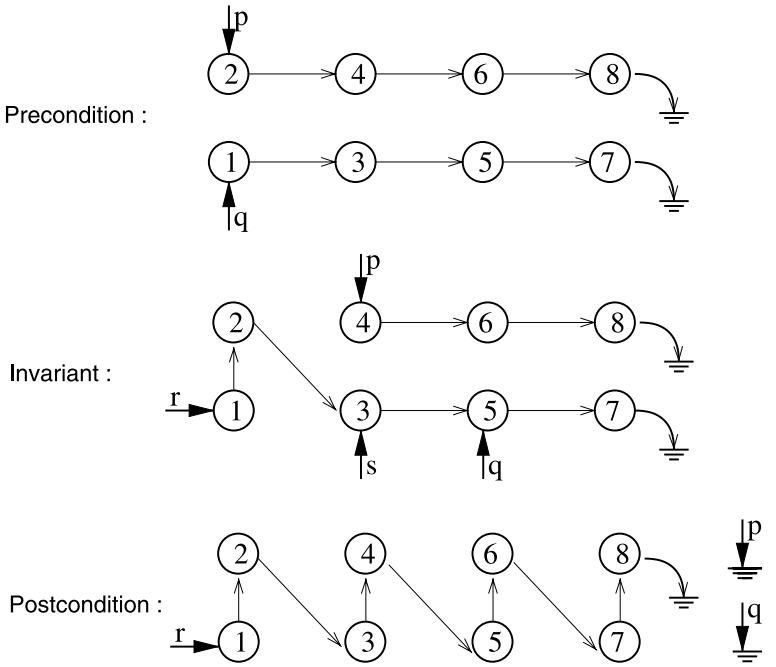

Fig. 2. In-place list merge.

$$Path :: next \Rightarrow ref \Rightarrow addr\ list \Rightarrow ref \Rightarrow bool$$

$$Path\ next\ x\ []\ y\ = (x = y) \quad [simp]$$
$$Path\ next\ x\ (a\#as)\ y = (x = Ref\ a \wedge Path\ next\ (next\ a)\ as\ y) \quad [simp]$$

This is a valid definition by primitive recursion on the list of addresses. Note that a path between any two locations need not be unique. Cyclic lists induce infinitely many paths, which will later prove problematic in the case of circular list reversal. But for the moment our definition of *Path* will do just fine.

Similar to our treatment for *List*, the following properties are proven using case distinction on *as* and are made available to Isabelle's automated proof tactics:

$$Path\ next\ Null\ as\ x = (as = [] \wedge x = Null) \quad [simp]$$
$$Path\ next\ (Ref\ a)\ as\ z = (\ as = [] \wedge z = Ref\ a\ \vee$$
$$(\exists\, bs.\ as = a\#bs \wedge Path\ next\ (next\ a)\ bs\ z)) \quad [simp]$$

As expected, a list in the heap is nothing but a path ending in *Null*:

$$Path\ next\ x\ as\ Null = List\ next\ x\ as$$

and the *separation lemma* for *Path* is analogous to the one for *List*

$$u \notin set\ as \Longrightarrow Path\ (next(u := v))\ x\ as\ y = Path\ next\ x\ as\ y \quad [iff]$$

Also, due to our rather liberal definition of *Path* we get a nice algebraic property:

$$Path\ next\ x\ (as\ @\ bs)\ z = (\exists\, y.\ Path\ next\ x\ as\ y \wedge Path\ next\ y\ bs\ z) \quad [simp]$$

In addition, a list can be decomposed into a path followed by a list:

$$List\ h\ x\ (as\ @\ bs) = (\exists\, y.\ Path\ h\ x\ as\ y \wedge List\ h\ y\ bs) \quad [simp]$$

We now return to our discussion of in-place list merge. Using *Path*, the loop invariant is:

$$\begin{array}{l} Merge\text{-}inv = \\ (\exists\ rs\ ps\ qs\ a. \\ Path\ next\ r\ rs\ s \wedge List\ next\ p\ ps \wedge List\ next\ q\ qs \wedge s = Ref\ a \wedge \\ merge(Ps,Qs,ord\ val) = rs\ @\ a\ \#\ merge(ps,qs,ord\ val) \wedge \\ (next\ a = p \vee next\ a = q) \wedge distinct\ (a\ \#\ ps\ @\ qs\ @\ rs)) \end{array}$$

In its present form, the proof of correctness requires user interaction in the form of interleaved invocations of automated proof tactics (the tableau prover combined with simplification) and instantiations of existential witnesses. The boolean condition of the *IF* statement is rewritten to its equivalent form in a BDD like representation in order to facilitate automatic case splitting. The following lemma is given to the tableau prover:

$$Path\ (next(a := q))\ p\ as\ (Ref\ a) \Longrightarrow Path\ (next(a := q))\ p\ (as\ @\ [a])\ q \quad [intro!]$$

The complete proof can be found online [10].

6.3. In-place circular list reversal

As alluded to earlier, in the case of cyclic lists, *Path* may traverse the cycle any number of times, and therefore allow for infinitely many paths from any location on the loop to another. We therefore define a new heap predicate *distPath* as follows:

$$distPath\ next\ x\ as\ y \equiv Path\ next\ x\ as\ y \wedge distinct\ as$$

The term *distPath next x as y* expresses the fact that a non-repeating path *as* connects location x to location y by means of the *next* field. In the case where $x = y$, and there is a cycle from x to itself, *as* can be both [] and the non-repeating list of nodes in the cycle. An example follows shortly.

We will now consider two algorithms for the reversal of circular lists. The first one, illustrated in Fig. 3 is specified as follows:

circular-list-rev-I:
VARS next root p q tmp
{*root* = *Ref r* ∧ *distPath next root* (*r*#*Ps*) *root*}
p := *root*; *q* := *root*ˆ.*next*;
WHILE q ≠ *root*
INV {∃ *ps qs*. *distPath next p ps root* ∧ *distPath next q qs root* ∧
root = *Ref r* ∧ *r* ∉ *set Ps* ∧ *set ps* ∩ *set qs* = {} ∧
Ps = (*rev ps*) @ *qs* }
DO tmp := *q*; *q* := *q*ˆ.*next*; *tmp*ˆ.*next* := *p*; *p*:=*tmp OD*;
*root*ˆ.*next* := *p*
{ *root* = *Ref r* ∧ *distPath next root* (*r*#*rev Ps*) *root*}

Referring to Fig. 3, in the beginning, we are able to assert *distPath next root as root*, with *as* set to [] or [*r*, *a*, *b*, *c*]. Note that *Path next root as root* would additionally give us an infinite number of lists with the recurring sequence [*r*, *a*, *b*, *c*].

The precondition states that there exists a non-empty non-repeating path *r* # *Ps* from pointer *root* to itself, given that *root* points to location *r*. Pointers *p* and *q* are then set to *root* and the successor of *root*, respectively. If *q* = *root*, we have circled the loop, otherwise we set the *next* pointer field of *q* to point to *p*, and shift *p* and *q* one step forward. The invariant thus states that *p* and *q* point to two disjoint lists *ps* and *qs*, such that *Ps* = *rev ps* @ *qs*. After the loop terminates, one extra

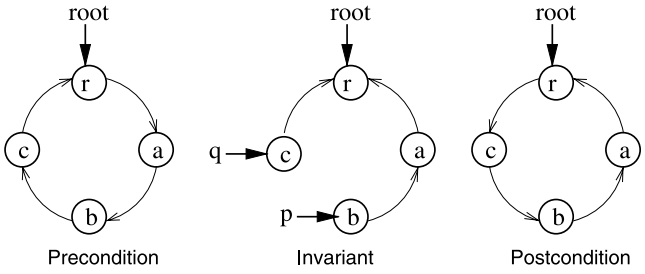

Fig. 3. Circular list reversal (I).

step is needed to close the loop. As expected, the postcondition states that the *distPath* from *root* to itself is now $r \# rev\ Ps$.

It may come as a surprise to the reader that the simple algorithm in Section 6.1 for acyclic list reversal, with modified annotations, works for cyclic lists as well:

circular-list-rev-II:
VARS next p q tmp
$\{p = Ref\ r \wedge distPath\ next\ p\ (r\#Ps)\ p\}$
q:=*Null*;
WHILE $p \neq Null$
INV
$\{ ((q = Null) \longrightarrow (\exists ps.\ distPath\ next\ p\ (ps)\ (Ref\ r) \wedge ps = r\#Ps)) \wedge$
$((q \neq Null) \longrightarrow (\exists ps\ qs.\ distPath\ next\ q\ (qs)\ (Ref\ r) \wedge List\ next\ p\ ps\ \wedge$
$set\ ps \cap set\ qs = \{\} \wedge rev\ qs\ @\ ps = Ps@[r])) \wedge$
$\neg (p = Null \wedge q = Null)\ \}$
DO tmp := *p*; *p* := *p*ˆ.*next*; *tmp*ˆ.*next* := *q*; *q*:=*tmp OD*
$\{q = Ref\ r \wedge distPath\ next\ q\ (r \#\ rev\ Ps)\ q\}$

Although the above algorithm is more succinct, its invariant is more involved. The reason for the case distinction on *q* is due to the fact that during execution, the pointer variables can point to either cyclic or acyclic structures. Fig. 4 illustrates this.

The proofs for both versions of circular list reversal are shorter, but about as automatic as the proof for *list-merge*. They comprise of alternating invocations of automated tactics, case splits, and instantiations of existential witnesses.

The following lemma is used by hand in the proof of *circular-list-rev-I*:

$[\![p \neq q;\ Path\ h\ p\ Ps\ q;\ distinct\ Ps]\!]$
$\Longrightarrow \exists a\ Qs.\ p = Ref\ a \wedge Ps = a \#\ Qs \wedge a \notin set\ Qs$

For *circular-list-rev-II*, an additional lemma relating *Path* and *List* is given as an introduction rule to the tableau prover:

$[\![Path\ next\ b\ Ps\ (Ref\ a);\ a \notin set\ Ps]\!] \Longrightarrow List\ (next(a := Null))\ b\ (Ps\ @\ [a])$

It states that if there exists a *Path* in between two points in a heap, a *List* can be obtained by setting the last point on the *Path* to *Null* in the heap. Complete proofs can be found online [10].

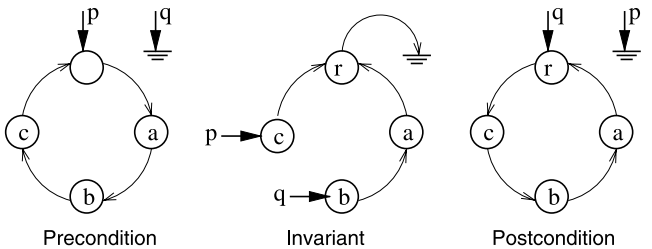

Fig. 4. Circular list reversal (II).

7. Extensions

As we have seen in the last section, verification proofs normally require manual instantiation of existential quantifiers. Although more powerful automatic provers for predicate calculus would help, for the moment we have found that providing a few witnesses interactively can be more economical than spending large amounts of time coaxing the system into finding a proof automatically.

Nevertheless, in the interest of more automatic proofs, this section starts with two approaches used to avoid existential quantifiers in loop invariants.

7.1. Functional abstraction

Trying to avoid existential quantifiers altogether, we resurrect our functional abstraction attempt in a logically sound way, using the *List* predicate:

$$\begin{array}{l} islist :: next \Rightarrow ref \Rightarrow bool \\ islist\ next\ p \equiv \exists as.\ List\ next\ p\ as \\ list :: next \Rightarrow ref \Rightarrow addr\ list \\ list\ next\ p \equiv SOME\ as.\ List\ next\ p\ as \end{array}$$

As a direct consequence we obtain:

$$List\ next\ p\ as = (islist\ next\ p \wedge as = list\ next\ p)$$

The following lemmas are easily derived from their counterparts for *List* and the relationship just proved:

$$\begin{array}{ll} islist\ next\ Null & [simp] \\ islist\ next\ (Ref\ a) = islist\ next\ (next\ a) & [simp] \\ list\ next\ Null = [] & [simp] \\ islist\ next\ (next\ a) \Longrightarrow list\ next\ (Ref\ a) = a\ \#\ list\ next\ (next\ a) & [simp] \\ islist\ next\ (next\ a) \Longrightarrow a \notin set\ (list\ next\ (next\ a)) & [simp] \\ [\![islist\ next\ p;\ y \notin set\ (list\ next\ p)]\!] \Longrightarrow islist\ (next(y := q))\ p & [simp] \\ [\![islist\ next\ p;\ y \notin set\ (list\ next\ p)]\!] \Longrightarrow list\ (next(y := q))\ p = list\ next\ p & [simp] \end{array}$$

This suffices for an automatic proof of list reversal:

$$\begin{array}{l} fun\text{-}abs\text{-}list\text{-}rev: \\ \quad VARS\ next\ p\ q\ r \\ \quad \{islist\ next\ p \wedge Ps = list\ next\ p\} \\ \quad q := Null; \\ \quad WHILE\ p \neq Null \\ \quad INV\ \{islist\ next\ p \wedge islist\ next\ q \wedge \\ \qquad set(list\ next\ p) \cap set(list\ next\ q) = \{\} \wedge \\ \qquad rev(list\ next\ p)\ @\ (list\ next\ q) = rev\ Ps\} \\ \quad DO\ r := p;\ p := p\hat{}.next;\ r\hat{}.next := q;\ q := r\ OD \\ \quad \{islist\ next\ q \wedge list\ next\ q = rev\ Ps\} \end{array}$$

Although both our linear list reversal proofs are automatic, the former (involving *List* and existential quantifiers) needs more search. From our experience with further proofs involving *islist* and *list*, we found that automation could eventually be achieved by proving further specialised rewrite rules. But this was less direct and more time consuming than providing existential witnesses for *List*. Thus we believe that relational abstraction, along with its associated existential quantification, is often easier to use than functional abstraction.

Functional abstraction is not possible for *Path* since it does not enjoy the uniqueness property that *List* has. It would be possible to do the same for *Path* if we were to redefine it to be unique by, for example, restricting its definition to allow only paths of minimal length.

7.2. Ghost variables

Another approach to tackling the problem of providing existential witnesses is to introduce separate new variables corresponding to the existentially quantified variables in the invariant. These variables play no part in the resulting computation, but act as aides in its proof. This is illustrated below for in-place list reversal:

$$
\begin{array}{l}
\textit{ghost-var-list-rev}: \\
\quad \textit{VARS next p q r ps qs} \\
\quad \{\textit{List next p Ps} \wedge \textit{ps} = \textit{Ps}\} \\
\quad \textit{q} := \textit{Null}; \textit{qs} := []; \\
\quad \textit{WHILE p} \neq \textit{Null} \\
\quad \textit{INV} \{\textit{List next p ps} \wedge \textit{List next q qs} \wedge \textit{set ps} \cap \textit{set qs} = \{\} \wedge \\
\qquad \textit{rev ps} \mathbin{@} \textit{qs} = \textit{rev Ps}\} \\
\quad \textit{DO} \\
\qquad \textit{r} := \textit{p}; \textit{p} := \textit{p}\hat{}.\textit{next}; \textit{r}\hat{}.\textit{next} := \textit{q}; \textit{q} := \textit{r}; \\
\qquad \textit{qs} := (\textit{hd ps}) \# \textit{qs}; \textit{ps} := \textit{tl ps} \\
\quad \textit{OD} \\
\quad \{\textit{List next q} (\textit{rev Ps})\}
\end{array}
$$

The original program variables have been augmented with *ghost variables*, *ps* and *qs* of type *addr list*, corresponding to the existentially quantified variables present in the loop invariant. Their initial values have been set in the precondition, and are updated at each loop iteration to reflect their current value. The proof is again automatic, but so were the proofs of the previous versions of list reversal.

For *list-merge* things improve: the introduction of ghost variables shortens the proof considerably and makes it almost automatic. This indicates that the main hurdle is the proper instantiation of existential witnesses.

Ghost variables eliminate the need for existential quantification, and therefore make proofs easier to automate. They also have the advantage that it is easier to think of how the contents of these variables change during execution, and come up with commands reflecting this, rather than coming up with existential witnesses during the proof process. The disadvantage of this approach is that entities from the logic (i.e., Isabelle lists, sets) enter the program text.

7.3. Other inductive data types on the heap

Till now, we have treated linked lists on the heap. A similar treatment is possible for other heap structures that correspond to inductively defined data types in our logic. The basic idea is simple: define the abstraction relation inductively, following the inductive definition of the data type. For instance, given the following data type for binary trees:

datatype $'a\ tree = Tip \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

the corresponding abstraction relation is defined recursively as:

$$\begin{aligned}
&Tree :: next \Rightarrow next \Rightarrow ref \Rightarrow addr\ tree \Rightarrow bool\\
&Tree\ l\ r\ p\ Tip \qquad\quad = (p = Null)\\
&Tree\ l\ r\ p\ (Node\ t1\ a\ t2) = (p = Ref\ a \wedge Tree\ l\ r\ (r\ a)\ t1 \wedge Tree\ l\ r\ (l\ a)\ t2)
\end{aligned}$$

Note that *Tree* actually characterises dags rather than trees. To avoid sharing we need an additional condition in the *Node*-case: $set\text{-}of\ t1 \cap set\text{-}of\ t2 = \{\}$ where *set-of* returns the nodes in a tree. Loops cannot arise because the definition of *Tree* is well-founded.

7.4. Modelling fault avoidance

What we have been considering so far is a Hoare logic for partial correctness (i.e., the post-condition holds at the end, if the precondition holds in the beginning *and* the program terminates normally). Under this interpretation, the following Hoare triple is trivially true:

$$\begin{aligned}
&VARS\ next\ p\\
&\{True\}\\
&p\hat{}.next := p\\
&\{Path\ next\ p\ []\ p\}
\end{aligned}$$

In cases where one would like the system to force the assertion $p \neq Null$ into the precondition, we need to model a fault-avoiding semantics. We extend our imperative programming language to include the command *Abort*, signalling abnormal termination of the program. We redefine our notion of partial correctness to include only normal termination (i.e., if the precondition holds, the program will not terminate abnormally).

We can now use *Abort* to signal a possible dereferencing of the *Null* pointer (i.e., a segmentation fault). The assignment $p\hat{}.next := p$ is internally translated into $IF\ p \neq Null\ THEN\ p\hat{}.next := p\ ELSE\ Abort\ FI$, making $p \neq Null$ the guard for the assignment to lead to normal termination, and therefore forcing it into the precondition:

$$\begin{aligned}
&VARS\ next\ p\\
&\{p \neq Null\}\\
&p\hat{}.next := p\\
&\{Path\ next\ p\ []\ p\}
\end{aligned}$$

We have thus treated the left-hand side of a pointer assignment. The treatment of a possible *Null* pointer dereference on the right-hand side of an assignment, and in program expressions in general, can be achieved using the techniques employed for fault avoidance, i.e., calculate guards for each

expression that ensure that they do not lead to a runtime fault, and augment the Hoare rules in order to take guards into account. Further details can be found elsewhere [18].

In the same way that the guard for the division operation $a \div b$ is $b \neq 0 \wedge g(a) \wedge g(b)$, the guard for the pointer dereference $p\hat{}.next$ is $p \neq Null \wedge g(p)$, where $g(p)$ calculates the guard for p in case of a multiple dereference. Such a check needs to be made for the left-hand side of an assignment as well.

Our experience with doing proofs in the abortive setting after their partial correctness proofs was that major parts of the original proofs could be reused since Isabelle's automatic tactics took care of most extra implications generated by the new guard conditions. Note that heap update $p\hat{}.next := p$ is now guarded against p being *Null*. However, p may still be illegal, uninitialised or dangling. To guard against that, one needs a more detailed model of the heap where allocated and free addresses are distinguished. This is needed anyway as soon as we want to reason about storage allocation/deallocation.

7.5. Storage allocation

We conclude our discussion by showing how we treat the allocation of new storage. Allocated addresses are distinguished from unallocated ones by introducing a separate variable that records the set of allocated addresses. Selecting a new address is easy:

$$new :: 'a\ set \Rightarrow 'a$$
$$new\ A \equiv SOME\ a.\ a \notin A$$

As long as the type of addresses is infinite (*UNIV* is the set of all elements of a given type) and the set of currently allocated addresses finite, a new address always exists:

$$[\![infinite\ UNIV; finite\ alloc]\!] \Longrightarrow new\ alloc \notin alloc$$

We can now introduce some syntactic sugar for memory allocation:

$$p := NEW\ alloc \quad = \quad p := Ref(new\ alloc); \quad alloc := \{addr\ p\} \cup alloc$$

Memory allocation is the sequential composition of generating a new address, and inserting this new address into the list of already allocated addresses.

We can now work on an algorithm for non-destructive linear list reversal. The following program copies a list in reverse order, i.e., creates a linked list on the heap whose *val* fields, in reverse order, are identical to the *val* fields of the input list pointed to by p. The term *map f xs* is the list obtained by applying the function f to each element of the list *xs*. Below, *map val* is used to map a list of addresses to the contents of their *val* fields.

non-dest-list-rev:
$\neg$ *finite* (*UNIV*::*addr set*) $\Longrightarrow$
VARS val next alloc t p (*q*::*ref*)
{*List next p Ps* $\wedge$ *set Ps* $\subseteq$ *alloc* $\wedge$ *finite alloc* $\wedge$ *Alloc* = *alloc*}
q := *Null*;
WHILE p $\neq$ *Null*

$$
\begin{array}{l}
\mathit{INV}\ \{ \\
\exists\ \mathit{ps}\ \mathit{qs}.\ \mathit{List}\ \mathit{next}\ p\ \mathit{ps} \wedge \mathit{List}\ \mathit{next}\ q\ \mathit{qs} \\
\quad \wedge\ \mathit{rev}\ (\mathit{map}\ \mathit{val}\ \mathit{qs})\ @\ \mathit{map}\ \mathit{val}\ \mathit{ps} = \mathit{map}\ \mathit{val}\ \mathit{Ps} \\
\quad \wedge\ \mathit{set}\ \mathit{ps} \subseteq \mathit{set}\ \mathit{Ps} \wedge \mathit{set}\ \mathit{Ps} \subseteq \mathit{Alloc} \\
\quad \wedge\ \mathit{finite}\ \mathit{alloc} \wedge \mathit{alloc} = \mathit{Alloc} \cup \mathit{set}\ \mathit{qs} \wedge \mathit{Alloc} \cap \mathit{set}\ \mathit{qs} = \{\} \\
\}\ \\
\mathit{DO}\ \\
\quad t := \mathit{NEW}\ \mathit{alloc}; \\
\quad t\hat{}.\mathit{next} := q;\ t\hat{}.\mathit{val} := p\hat{}.\mathit{val};\ q := t;\ p := p\hat{}.\mathit{next} \\
\mathit{OD}; \\
p := r \\
\{\exists\ \mathit{Qs}.\ \mathit{List}\ \mathit{next}\ q\ \mathit{Qs} \wedge\ \mathit{map}\ \mathit{val}\ \mathit{Qs} = \mathit{rev}\ (\mathit{map}\ \mathit{val}\ \mathit{Ps}) \\
\qquad \wedge\ \mathit{alloc} = \mathit{Alloc} \cup \mathit{set}\ \mathit{Qs} \wedge \mathit{Alloc} \cap \mathit{set}\ \mathit{Qs} = \{\}\ \}
\end{array}
$$

We assume that the type of addresses is infinite. Program variable *alloc* contains the set of allocated addresses. Auxiliary variable *Alloc* is the set of initially allocated addresses. The proof of the algorithm is not automatic.

Although treating storage allocation in this way is possible in principle, dragging around the set *Alloc* may prove inconvenient in practice. A more natural treatment of storage allocation and deallocation can be found in separation logic [17].

8. The Schorr–Waite algorithm

This section demonstrates that our model and implementation can deal with a non-trivial algorithm that is considered a benchmark for pointer formalisations, the Schorr–Waite algorithm. To allow comparison with existing work we do not invent yet another proof of this algorithm. We intentionally pick an existing proof, one that has been mechanised before, again by Bornat, whose invariants we take over pretty much unchanged. Instead, and this is the second purpose of this section, we want to show that our proof is not just shorter than Bornat's but also almost human readable. Hence we present major parts of it, interspersed with explanation of the notation.

It must be emphasised that the point of presenting the proof in such detail is not because the proof is of intrinsic interest, but because we want to demonstrate that a general purpose formal proof language is also suitable for program proofs. Why? Because we do not believe that full-blown program verification will ever become completely automatic. Therefore our aim must be to provide (on top of as much automation as possible) a formal proof language that is not just checkable for the computer but also readable for the human. At the moment, only the interactive theorem provers Mizar and Isabelle provide such languages, but neither system has been used for the verification of a non-trivial imperative algorithm, let alone one using pointers. In that sense our proof is a true novelty.

However, it will become clear when we go through the proof, that *Isar*, Isabelle's structured proof language, is not immediately readable without further explanations (which we provide). It shares this fate with most other formal languages. There will also be points where the formal proof appears unnecessarily long (and others where it is surprisingly short). Hence the detailed exposition of this proof is primarily a presentation of the state of the art in structured program proofs, a point of reference for future efforts in this direction, and above all a motivation to narrow the gap between

machine supported proofs and detailed journal-style proofs further. We strongly believe that this gap is going to disappear altogether in the long run.

8.1. The algorithm

The Schorr–Waite algorithm [19] is a non-recursive graph marking algorithm. Most graph marking algorithms (e.g., depth-first or breadth-first search) are recursive, making their proof of correctness relatively simple. In general one can eliminate recursion in favour of an explicit stack. In certain cases, the need for an explicit stack can be relaxed by using the data structure at hand to store state information. The Schorr–Waite algorithm does just that. The incentive for this is not merely academic. Graph marking algorithms are normally used during the first stage of garbage collection, when scarcity of memory prohibits the luxury of a stack.

The problem with graph marking without recursion is backtracking: we have to remember where we came from. The Schorr-Waite algorithm uses the fact that if we always keep track of the current predecessor node once we have descended into the next node in the graph, the pointer reference from the predecessor to the next node is redundant, and can be put to better use by having it point to the predecessor of this predecessor node, and so on till the root of the graph. If done carefully, this reverse pointer chain preserves connectivity, facilitates backtracking through the graph, and is analogous to a stack.

Fig. 5 illustrates a complete marking cycle for a small subgraph. We have a pointer to the next node to be considered (i.e., the *tip*, t) and to its previously visited predecessor (p). The tip is marked and the algorithm descends into its left child, updating the predecessor pointer, and using the forward link of the tip to point to its predecessor. The tip has been "pushed" onto the predecessor stack. After exploring the left child, a "swing" is performed to do the same with the right. When all children of our original tip have been explored, no more swings are possible, and the tip is "popped" out of the predecessor stack, leaving us with the original subgraph with all reachable nodes marked.

Every pointer that is traversed in the graph is reversed, making it non-trivial to see that we are indeed left with the graph we had started with, when the algorithm has terminated. This difficulty is amplified when one tries to formally prove its correctness. The Schorr–Waite algorithm is therefore considered a benchmark for any pointer formalisation. Below is the version of the algorithm we will prove correct in this paper, along with Hoare logic assertions which we will discuss in the next section.

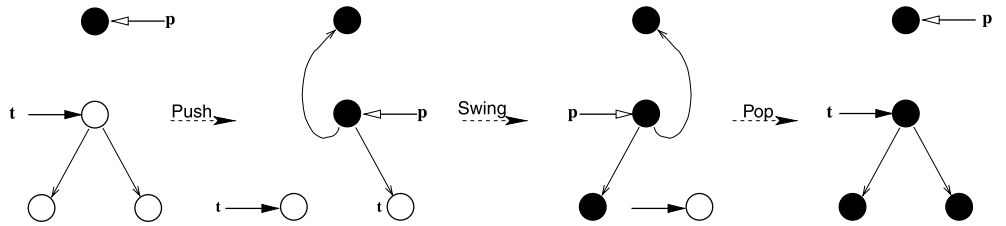

Fig. 5. A marking cycle.

SchorrWaite-I:
VARS c m l r t p q root
$\{R = reachable\ (relS\ \{l, r\})\ \{root\} \wedge (\forall x.\ \neg\ m\ x) \wedge iR = r \wedge iL = l\}$
$t := root;\ p := Null;$
WHILE $p \neq Null \vee t \neq Null \wedge \neg\ t\hat{}.m$
INV $\{\exists\ stack.$
$List\ (S\ c\ l\ r)\ p\ stack \wedge$ $(*i1*)$
$(\forall x \in set\ stack.\ m\ x) \wedge$ $(*i2*)$
$R = reachable\ (relS\{l, r\})\ \{t,p\} \wedge$ $(*i3*)$
$(\forall x.\ x \in R \wedge \neg m\ x \longrightarrow$ $(*i4*)$
$x \in reachable\ (relS\{l,r\}|m)\ (\{t\} \cup set(map\ r\ stack))) \wedge$
$(\forall x.\ m\ x \longrightarrow x \in R) \wedge$ $(*i5*)$
$(\forall x.\ x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge$ $(*i6*)$
$(stkOk\ c\ l\ r\ iL\ iR\ t\ stack)$ $(*i7*)\ \}$
DO
IF $t = Null \vee t\hat{}.m$
THEN
IF $p\hat{}.c$
THEN $q := t;\ t := p;\ p := p\hat{}.r;\ t\hat{}.r := q$ $(*pop*)$
ELSE $q := t;\ t := p\hat{}.r;\ p\hat{}.r := p\hat{}.l;$ $(*swing*)$
$p\hat{}.l := q;\ p\hat{}.c := True$
FI
ELSE $q := p;\ p := t;\ t := t\hat{}.l;\ p\hat{}.l := q;$ $(*push*)$
$p\hat{}.m := True;\ p\hat{}.c := False$
FI
OD
$\{(\forall x.\ (x \in R) = m\ x) \wedge (r = iR \wedge l = iL)\ \}$

We consider graphs where every node has at most two successors. The proof with arbitrary out degree uses the same principles and is just a bit more tedious. For every node in the graph, l and r are pointer fields that point to the successor nodes, m is a boolean field that is true for all marked nodes, and will be the result of running the algorithm. The boolean helper field c keeps track of which of the two child pointers has been reversed. Pointer t points to the tip. It is initially set to the *root*. Within the while loop, the algorithm divides into three arms, corresponding to the operation being performed on the predecessor stack. Pointer p points to the predecessor of t and is also the top of the predecessor stack.

8.2. Specification

The specification uses the following auxiliary definitions:

$reachable\ r\ P \equiv r^*\ ``\ addrS\ P$ (*reachable-def*)
$addrS\ P \equiv \{a.\ Ref\ a \in P\}$ (*addrS-def*)
$relS\ M \equiv \bigcup_{m \in M} \{(a, b).\ m\ a = Ref\ b\}$ (*relS-def*)
$r \mid m \equiv \{(x, y).\ (x, y) \in r \wedge \neg\ m\ x\}$ (*restr-def*)

Reachability is defined as the image of a set of addresses under a relation (r `` S is the image of set S under relation r). This relation is given by *relS* which casts a set of mappings (i.e., field names) to a relation. $r|m$ is the restriction of the relation r w.r.t. the boolean mapping m.

We will now explain the Hoare logic assertions shown in Section 8. The precondition requires all nodes to be unmarked. It "remembers" the initial value of l, r and the set of nodes reachable from *root* in *iL*, *iR* and *R*, respectively. As the postcondition we want to prove that a node is marked iff it is in *R*, i.e., is reachable, and that the graph structure is unchanged. To prove termination, we would need to show that there exists a loop measure that decreases with each iteration. Bornat [3] points out a possible loop measure. Since our Hoare logic implementation does not deal with termination, we prove only partial correctness.

The loop invariant is a bit more involved. Every time we enter the loop, *stack* is made up of the list of predecessor nodes starting at p, using the mapping $S\ c\ l\ r \equiv \lambda x.\ if\ c\ x\ then\ r\ x\ else\ l\ x$, that returns l or r depending on the value of c (*i1*). Everything on the stack is marked (*i2*). Everything initially reachable from *root* is now reachable from t and p (*i3*). If something is reachable and unmarked, it is reachable using only unmarked nodes from t or from the r fields of nodes in the stack (we traverse l before r) (*i4*). If a node is marked, it was initially reachable (*i5*). All nodes not on the stack have their l and r fields unchanged (*i6*). *stkOk* says that for the nodes on the stack we can reconstruct their original l and r fields (*i7*). It is defined using primitive recursion:

$$stkOk\ c\ l\ r\ iL\ iR\ t\ [] = True \qquad [simp]$$

$$stkOk\ c\ l\ r\ iL\ iR\ t\ (p\#stk) = (stkOk\ c\ l\ r\ iL\ iR\ (Ref\ p)\ (stk) \wedge iL\ p = (if\ c\ p\ then\ l\ p\ else\ t) \wedge iR\ p = (if\ c\ p\ then\ t\ else\ r\ p) \qquad [simp]$$

8.3. Proof of correctness

In this section we will go through part of the Isabelle/Isar proof of correctness, emphasising its readability. Although we provide additional comments, we rely on the self-explanatory nature of the Isar proof language, details of which can be found elsewhere [21,11]. The entire proof is available at [10]. At many places in the proof a compromise was made between using automatic proof tactics when the proof looked intuitive, and manually going into the proof state when it was felt that more explanation was necessary. The entire proof is about 400 lines of text. As far as we know, it is the shortest and most human readable, machine checkable proof of this algorithm. References to traditional proofs can be found in [3].

As described in Section 2.2, for automatic proofs, Isabelle is equipped with a number of proof methods. In the actual proof text, *simp* refers to the simplifier, *blast* to the pure tableau prover, and *auto* is mostly simplification with just a little proof search. Sometimes we do not show the actual proof method but only the additional lemmas fed to it. The omitted proof method is always the combination of tableau search and simplification.

For every construct defined, we prove its corresponding *separation lemmas*, such as the one for *List* in Section 5.2. They are used as simplification rules wherever applicable. Proofs of these *separation lemmas* normally follow from short and simple inductive arguments. The complete proof document [10] contains all such proven simplification rules.

We first state the correctness theorem as the Hoare triple in Section 8 and use the Isabelle verification condition generator *vcg* to reduce it to pure HOL subgoals.

Note that assertions are modelled as functions that depend on program variables. This is a standard way of modelling the dependencies of an assertion in a higher-order setting. Substitution in an assertion is therefore simply function application with changed parameters.

We abbreviate the loop invariant to *?inv*. The *?* before *inv* denotes that it is a schematic variable. Schematic variables are merely abbreviations for other terms. Since the loop invariant depends on the program variables *c l m r t p*, *?inv* is actually a function, and it is *?inv c l m r t p* that represents the invariant as it appears in *SchorrWaite-I*. The definition of *?inv* is done using the **let** command, but has been omitted. More examples with **let** follow shortly.

Precondition leads to invariant

We first show that the precondition leads to the invariant. Starting from the precondition, we need to prove *?inv c l m r root Null* (i.e., *?inv c l m r t p* pulled back over the initial assignments $t :=$ *root*; $p :=$ *Null*). In our goal, since $p =$ *Null*, the variable *stack* under the existential is the empty list. This simplifies things sufficiently, making the proof trivial. (Note: **fix** introduces new free variables into a proof – the statement is proved for "arbitrary but fixed values".)

fix *c m l r t p q root*
assume *?Pre c m l r root*
thus *?inv c m l r root Null* **by** *auto*
next

Invariant and loop termination imply postcondition

Next we show the postcondition to be true, given that the invariant and loop termination condition hold. By pattern matching, we first get hold of the formula *?Inv* under the existential quantifier of the invariant. A new variable *stack* is introduced as an existential witness and *?Inv stack* is decomposed into its smaller conjuncts, *?I1*, *?I4* etc. that are relevant to the proof.

Command **let** matches the pattern on the left with the term on the right, instantiating all *?*-variables in the process. Note that *?I1*, etc., are merely formulae, i.e., syntax, and that the corresponding facts *i1*, etc., need to be proven explicitly (from *inv* using $\wedge$-elimination). Intuitively, the comments $((*i1*)\ldots(*i7*))$ in the statement of *SchorrWaite-I* correspond to the schematic variables (*?I1* ... *?I7*), and the proven facts (*i1* ... *i7*).

fix *c m l r t p q*
let $\exists$ *stack*. *?Inv stack* = *?inv c m l r t p*
assume *a*: *?inv c m l r t p* $\wedge \neg(p \neq Null \vee t \neq Null \wedge \neg\, t\hat{}.m)$
then obtain *stack* **where** *inv*: *?Inv stack* **by** *blast*
from *a* **have** *pNull*: $p = Null$ **and** *tDisj*: $t{=}Null \vee (t{\neq}Null \wedge t\hat{}.m\ )$ **by** *auto*
let *?I1* $\wedge$ - $\wedge$ - $\wedge$ *?I4* $\wedge$ *?I5* $\wedge$ *?I6* $\wedge$ - = *?Inv stack*
from *inv* **have** *i1*: *?I1* **and** *i4*: *?I4* **and** *i5*: *?I5* **and** *i6*: *?I6* **by** *auto*

Using $p =$ *Null*, *stack* is then shown to be the empty list:

from *pNull i1* **have** *stackEmpty*: *stack* = [] **by** *simp*

The only meaty part of this proof is to show that all nodes in R are marked. This is shown using the previously proven case distinction on t, followed by a contradiction argument using *i4*.

from *tDisj i4* **have** *RisMarked*: $\forall x.\ x \in R \longrightarrow m\ x$
by *reachable-def stackEmpty*

The first half of the postcondition follows from *i6* and *RisMarked*. Since *stack* is empty, the fact that all *l* and *r* mappings are restored follows directly from *i5*, which completes the subproof:

from *i5 i6* **show** $(\forall x.(x \in R) = m\ x) \wedge r = iR \wedge l = iL$
by *stackEmpty RisMarked*
next

Invariant is preserved

The bulk of the proof lies in trying to prove that the invariant is preserved. Assuming the invariant and loop condition hold, we need to show the invariant after variable substitution arising from all three arms of the algorithm. After a case distinction on the if-then-else conditions we are left with three large but similar subproofs. In this paper we will only walk through the proof of the pop arm in order to save whatever is left of the reader's interest. The pop arm serves as a good illustration as it involves the "seeing is believing" graph reconstruction step, a decrease in the length of the stack, as well as a change of the graph mapping *r*.

fix *c m l r t p q root*
let $\exists$ *stack*. *?Inv stack* = *?inv c m l r t p*
let $\exists$ *stack*. *?popInv stack* = *?inv c m l* $(r(p \rightarrow t))$ *p* $(p\hat{}.r)$
assume $(\exists$ *stack*.*?Inv stack*$) \wedge (p \neq Null \vee t \neq Null \wedge \neg\ t\hat{}.m)$
(**is** - $\wedge$ *?whileB*)
then obtain *stack* **where** *inv*: *?Inv stack* **and** *whileB*: *?whileB* **by** *blast*
let $?I1 \wedge ?I2 \wedge ?I3 \wedge ?I4 \wedge ?I5 \wedge ?I6 \wedge ?I7 = ?Inv\ stack$
from *inv* **have** *i1*: *?I1* **and** *i2*: *?I2* **and** *i3*: *?I3* **and** *i4*: *?I4*
and *i5*: *?I5* **and** *i6*: *?I6* **and** *i7*: *?I7* **by** *auto*

We start by dismantling the invariant and pattern matching its seven conjuncts with *?I1* ... *?I7*. Command **is**, like **let** performs pattern matching. *?Inv* is the original invariant after existential elimination using the witness *stack*. *?popInv* corresponds to *?Inv* pulled back over the pop arm assignments.

We begin the pop arm proof by assuming the two if-then-else conditions and proving facts that we use later. We introduce a new variable *stack-tl* to serve as the witness for $\exists$ *stack*. *?popInv stack*, our goal.

assume *ifB1*: $t = Null \vee t\hat{}.m$ **and** *ifB2*: $p\hat{}.c$
from *ifB1 whileB* **have** *pNotNull*: $p \neq Null$ **by** *auto*
then obtain *addr-p* **where** *addr-p-eq*: $p = Ref\ addr\text{-}p$ **by** *auto*
with *i1* **obtain** *stack-tl* **where** *stack-eq*: $stack = (addr\ p)\ \#\ stack\text{-}tl$
by *auto*
with *i2* **have** *m-addr-p*: $p\hat{}.m$ **by** *auto*
have *stackDist*: *distinct* (*stack*) **using** *i1* **by** (*rule List-distinct*)
from *stack-eq stackDist* **have** *p-notin-stack-tl*:$addr\ p \notin set\ stack\text{-}tl$
by *simp*

We now prove the seven individual conjuncts of *?popInv stack-tl* separately as facts *poI1* to *poI7*, which we state explicitly. Note that we could also pattern match *?popInv stack-tl* to assign these indi-

vidual conjuncts to seven *?*-variables, eliminating the need to mention them explicitly. In general, it is a good idea to instantiate *?*-variables to use later in proofs. Like user defined constants in programs, it makes proofs a lot more tolerant to change and allows one to see their structure. The disadvantage is that too much pattern matching and back referencing makes the proof difficult to read.

Our first goal follows directly from our assumptions and definitions. The *separation lemma* for *List* is used:

— List property is maintained:
from *i1 p-notin-stack-tl ifB2*
have *poI1*: *List* (*S c l* (*r*(*p* → *t*))) (*p*^.*r*) *stack-tl*
by *addr-p-eq stack-eq S-def*
moreover

Next we have to show that all nodes in *stack-tl* are marked. This follows directly from our original invariant, where we know that all nodes in *stack* are marked.

— Everything on the stack is marked:
from *i2* **have** *poI2*: ∀ *x* ∈ *set stack-tl*. *m x* **by** (*simp add*:*stack-eq*)
moreover

Next we prove that all nodes are still reachable after executing the pop arm. We need the help of lemma *still-reachable* that we have proven separately:

$$[\![B \subseteq Ra^* \mathbin{``} A;\ \forall (x,y) \in Rb - Ra.\ y \in Ra^* \mathbin{``} A]\!] \Longrightarrow Rb^* \mathbin{``} B \subseteq Ra^* \mathbin{``} A$$

A little pattern matching will give us something of the form to which we can apply this lemma.

— Everything is still reachable:
let (*R* = *reachable ?Ra ?A*) = *?I3*
let *?Rb* = (*relS* {*l*, *r*(*p* → *t*)})
let *?B* = {*p*, *p*^.*r*}
— Our goal is *R* = *reachable ?Rb ?B*.
have *?Ra** `` *addrS ?A* = *?Rb*\* `` *addrS ?B* (**is** *?L* = *?R*)
proof
show *?L* ⊆ *?R*
proof (*rule still-reachable*)
show *addrS ?A* ⊆ *?Rb** `` *addrS ?B* **by** *relS-def oneStep-reachable*

After filling in the pattern matched variables, this last subgoal is:

addrS {*t*, *p*} ⊆ (*relS* {*l*, *r*(*p* → *t*)})* `` *addrS* {*p*, *p*^.*r*}

and is true as *p* can be reached by reflexivity, and *t* by a one step hop from *p*. The second subgoal generated by *still-reachable* is:

∀ (*x*, *y*)∈*relS* {*l*, *r*} − *relS* {*l*, *r*(*p* → *t*)}.
y ∈ (*relS* {*l*, *r*(*p* → *t*)})* `` *addrS* {*p*, *p*^.*r*}

and can be seen to be true as if any such pair (*x*,*y*) exists, it has to be (*p*, *p*^.*r*):

show ∀ (*x*,*y*) ∈ *?Ra*−*?Rb*. *y* ∈ (*?Rb** `` *addrS ?B*)
by *relS-def addrS-def*
qed

The other direction of $?L = ?R$ can be shown to be correct by similar arguments and is proven by appropriately instantiated automatic proof tactics.

show $?R \subseteq ?L$ — Proof hidden
qed
with *i3* **have** *poI3*: $R = reachable\ ?Rb\ ?B$ **by** (*simp add:reachable-def*)
moreover

The proof for the next part of the invariant is a bit more indirect.

— If it is reachable and not marked, it is still reachable using...
let $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Ra\ ?A\ =\ ?I4$
let $?Rb = relS\ \{l, r(p \rightarrow t)\} \mid m$
let $?B = \{p\} \cup set\ (map\ (r(p \rightarrow t))\ stack\text{-}tl)$
— Our goal is $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Rb\ ?B$.
let $?T = \{t, p\hat{}.r\}$

Assuming we have an x that satisfies $x \in R \wedge \neg\ m\ x$, we have $x \in reachable\ ?Ra\ ?A$ (from *i4*). What we need is $x \in reachable\ ?Ra\ ?B$. Examining these two sets, we see that their difference is $reachable\ ?Rb\ ?T$, which is the set of elements removed from $reachable\ ?Ra\ ?A$ as a result of the pop arm. We therefore do the proof in two stages. First we prove the subset with difference property, and then show that this fits with what happens in the pop arm.

have $?Ra^*\ ``\ addrS\ ?A \subseteq ?Rb^*\ ``\ (addrS\ ?B \cup addrS\ ?T)$
— Proof hidden; similar to previous use of *still-reachable*
— We now bring a term from the right to the left of the subset relation.
hence *subset*: $?Ra^*\ ``\ addrS\ ?A - ?Rb^*\ ``\ addrS\ ?T \subseteq ?Rb^*\ ``\ addrS\ ?B$
by *blast*
have *poI4*: $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Rb\ ?B$
proof
fix x **assume** *a*: $x \in R \wedge \neg\ m\ x$
— First, a disjunction on $p\hat{}.r$ used later in the proof
have *pDisj*:$p\hat{}.r = Null \vee (p\hat{}.r \neq Null \wedge p\hat{}.r\hat{}.m)$ **using** *poI1 poI2*
by *auto*
— x belongs to the left hand side of *subset*:
have *incl*: $x \in ?Ra^*``addrS\ ?A$ **using** *a i4* **by** *reachable-def*
have *excl*: $x \notin ?Rb^*``addrS\ ?T$ **using** *pDisj ifB1 a* **by** *addrS-def*
— And therefore also belongs to the right hand side of *subset*,
— which corresponds to our goal.
from *incl excl subset* **show** $x \in reachable\ ?Rb\ ?B$ **by** *reachable-def*
qed
moreover

Since m is unchanged through the pop arm, the next subgoal is identical to its counterpart in the original invariant.

— If it is marked, then it is reachable
from *i5* **have** *poI5*: $\forall x.\ m\ x \longrightarrow x \in R$ **.**
moreover

The next part of the invariant is what is used to prove that the *l* and *r* are finally restored. As expected, the major part of this proof follows from *i7*, the assertion involving *stkOk*, expressing what it means for a graph to be reconstructible.

— If it is not on the stack, then its *l* and *r* fields are unchanged
from *i7 i6 ifB2*
have *poI6*: $\forall x.\ x \notin set\ stack\text{-}tl \longrightarrow (r(p \rightarrow t))\ x = iR\ x \wedge l\ x = iL\ x$
by *addr-p-eq stack-eq*
moreover

The last part of the invariant involves the *stkOk* predicate. The only thing the pop arm changes here is the *r* mapping at *p*. The goal is automatically proven using the following simplification rule:

$x \notin set\ xs \Longrightarrow$
$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$

— If it is on the stack, then its *l* and *r* fields can be reconstructed
from *p-notin-stack-tl i7* **have** *poI7*: $stkOk\ c\ l\ (r(p \rightarrow t))\ iL\ iR\ p\ stack\text{-}tl$
by *stack-eq addr-p-eq*

The proof of the pop arm was in the style of an Isabelle "calculation," with **have** statements separated by **moreover**, which can **ultimately** be put together to show the goal at hand. At this point we have proved the individual conjuncts of *?popInv stack-tl*. We will now piece them together and introduce an existential quantifier, thus arriving exactly at what came out of the verification condition generator:

ultimately show *?popInv stack-tl* **by** *simp*
qed
hence $\exists$ *stack*. *?popInv stack* **..**

We similarly prove preservation of the invariant in the swing and push arms and combine these results to complete the proof.

8.4. A second graph marking algorithm

To test the reusability of the above proof, we have verified a second graph marking algorithm. Below is a graph marking algorithm also using the Schorr–Waite strategy. It is developed [1] in Atelier B [7] with the Click'n Prove [2] interface. Its development heavily uses the refinement technique.

SchorrWaite-II:
VARS fin c m l r t p q root
$\{R = reachable\ (relS\ \{l, r\})\ \{root\} \wedge (\forall x.\ \neg\ m\ x) \wedge$
$iR = r \wedge iL = l \wedge root \neq Null\}$
$t := root;\ p := Null;\ fin := False;\ t\hat{}.m := True;$
WHILE $\neg fin$
INV $\{\ \exists stack.$
$(t \neq Null \wedge t\hat{}.m \wedge addr\ t \notin set\ stack$
$\wedge\ (fin \longrightarrow stack = [] \wedge (\forall x.\ x \in R \longrightarrow m\ x))) \wedge$ $(*i0*)$

$$
\begin{array}{ll}
\quad List\ (S\ c\ l\ r)\ p\ stack \wedge & (*i1*)\\
\quad (\forall x \in set\ stack.\ m\ x) \wedge & (*i2*)\\
\quad R = reachable\ (relS\{l, r\})\ \{t,p\} \wedge & (*i3*)\\
\quad (\forall x.\ x \in R \wedge \neg m\ x \longrightarrow x \in reachable\ (relS\{l,r\}|m) & \\
\qquad (\{(t\hat{}.l),(t\hat{}.r)\} \cup set(map\ r\ stack))) \wedge & (*i4*)\\
\quad (\forall x.\ m\ x \longrightarrow x \in R) \wedge & (*i5*)\\
\quad (\forall x.\ x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge & (*i6*)\\
\quad (stkOk\ c\ l\ r\ iL\ iR\ t\ stack) & (*i7*)
\end{array}
$$

$$
\begin{array}{l}
\}\\
DO\\
\ IF\ (t\hat{}.l \neq Null \wedge \neg(t\hat{}.l)\hat{}.m)\ THEN\\
\ \ (t\hat{}.l)\hat{}.m := True;\ t\hat{}.c := False;\ q{:=}p;\ p{:=}t;\ t{:=}\ t\hat{}.l;\ p\hat{}.l{:=}q\quad (*downL*)\\
\ ELSE\ IF\ (t\hat{}.r \neq Null \wedge \neg(t\hat{}.r)\hat{}.m)\ \ THEN\\
\ \ (t\hat{}.r)\hat{}.m := True;\ t\hat{}.c := True;\ \ q{:=}p;\ p{:=}t;\ t{:=}\ t\hat{}.r;\ p\hat{}.r{:=}q\ \ (*downR*)\\
\ ELSE\ IF\ (p \neq Null)\ THEN\\
\ \ \ IF\ (p\hat{}.c)\\
\ \ \ THEN\ \ q{:=}p;\ p{:=}p\hat{}.r;\ q\hat{}.r{:=}t;\ t{:=}q\quad\ (*upR*)\\
\ \ \ ELSE\ \ q{:=}p;\ p{:=}p\hat{}.l;\ q\hat{}.l{:=}t;\ t{:=}q\quad\ (*upL*)\\
\ \ \ FI\\
\ ELSE\ \ fin := True\\
\ FI\ FI\ FI\\
OD\\
\{(\forall x.\ (x \in R) = m\ x) \wedge (r = iR \wedge l = iL)\ \}
\end{array}
$$

It differs from the previous version of Schorr–Waite in the following ways:

- The tip t is always marked.
- The boolean variable *fin* denotes termination of the algorithm.
- The loop splits into two main arms: *down*, going one level deeper in the graph, and *up*, a backtracking step. Each main arm then divides into two very symmetric sub-arms, one for each of the pointer fields l, r.
- Each node is visited once, plus the number of its currently unmarked child nodes. The previous algorithm visits each marked node thrice, making it somewhat less efficient.

Despite these differences, the proof of *SchorrWaite-II* uses the same principles as that *SchorrWaite-I*. The loop invariant needs to be augmented with some particularities of this algorithm (*i0*), and the property *i4* needs to be modified for the tip always being marked.

Although we do not present any part of the proof here, it is largely composed of re-instantiated chunks of the previous proof, made possible by the modular nature of the Isar proof language. The complete proof is available on the author's homepage [10].

9. Conclusion

We have presented a logically sound approach to reasoning about pointer algorithms using Hoare logic in Isabelle/HOL. Various verification examples using linked lists (acyclic and cyclic)

have been treated in detail, along with possible methods for automating their proofs. Additionally, we have touched on the subjects of storage allocation and Hoare logic with an abortive semantics.

Our main case study, a verification of the Schorr–Waite graph marking algorithm, illustrates the effectiveness of our approach for complicated examples. The proof presented in this paper is the shortest machine checkable, yet still human readable proof of this algorithm that we are aware of.

So what about a fully automatic proof of the Schorr–Waite algorithm? This seems feasible: once the relevant inductive lemmas are provided, the preservation of the invariant in the algorithm should be reducible to a first-order problem (with some work, as we currently employ higher-order functions). If the proof is within reach of current automatic first-order provers is another matter that we intend to investigate in the future. But irrespective of that, a readable formal proof is of independent interest because the algorithm is sufficiently complicated that a mere "yes, it works" is not satisfactory.

References

[1] J.R. Abrial, Event based sequential program development: application to constructing a pointer program, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), Formal Methods Europe (FME 2003), Lect. Notes in Comp. Sci., vol. 2805, Springer-Verlag, Berlin, 2003, pp. 51–74.

[2] J.R. Abrial, D. Cansell, Click'n prove: interactive proofs within set theory, in: D. Basin, B. Wolff (Eds.), Theorem Proving in Higher Order Logics, TPHOLs 2003, Lect. Notes in Comp. Sci., vol. 2758, Springer-Verlag, Berlin, 2003, pp. 1–24.

[3] R. Bornat, Proving pointer programs in Hoare Logic, in: R. Backhouse, J. Oliveira (Eds.), Mathematics of Program Construction (MPC 2000), Lect. Notes in Comp. Sci., vol. 1837, Springer-Verlag, Berlin, 2000, pp. 102–126.

[4] R. Bornat, B. Sufrin, Animating formal proofs at the surface: the Jape proof calculator, Comput. J. 43 (1999) 177–192.

[5] R. Bornat, Proofs of pointer programs in Jape. Available from: <http://www.dcs.qmul.ac.uk/~richard/pointers/>.

[6] R. Burstall, Some techniques for proving correctness of programs which alter data structures, in: B. Meltzer, D. Michie (Eds.), Machine Intelligence 7, Edinburgh University Press, 1972, pp. 23–50.

[7] Clearsy, Atelier B. User Manual, Aix-en-Provence (2001).

[8] M.J.C. Gordon, Mechanizing programming logics in higher order logic, Curr. Trends Hardware Verification Automated Theorem Proving (1989) 387–439.

[9] J.L. Jensen, M.E. Joergensen, M.I. Schwartzbach, N. Klarlund, Automatic verification of pointer programs using monadic second-order logic, in: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, ACM Press, 1997, pp. 226–234.

[10] F. Mehta, T. Nipkow, Proving pointer programs in higher-order logic. Available from: <http://www.in.tum.de/~nipkow/pubs/iandc05.html>.

[11] T. Nipkow, Structured proofs in Isar/HOL, in: H. Geuvers, F. Wiedijk (Eds.), Types for Proofs and Programs (TYPES 2002), Lect. Notes in Comp. Sci., vol. 2646, Springer-Verlag, Berlin, 2003, pp. 259–278.

[12] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, Lect. Notes in Comp. Sci., vol. 2283 , Springer-Verlag, Berlin, 2002. Available from: http://www.in.tum.de/~nipkow/LNCS2283/.

[13] T. Nipkow, Winskel is (almost) right: towards a mechanized semantics textbook, Formal Asp. Comput. 10 (1998) 171–186.

[14] L.C. Paulson, Generic automatic proof tools, in: R. Veroff (Ed.), Automated Reasoning and its Applications: Essays in Honor of Larry Wos, MIT Press, 1997, pp. 23–47.

[15] L.C. Paulson, A generic tableau prover and its integration with Isabelle, Tech. Rep. 441, University of Cambridge, Computer Laboratory, 1998.

[16] J.C. Reynolds, Intuitionistic reasoning about shared mutable data structures, in: J. Davies, B. Roscoe, J. Woodcock (Eds.), Millenial Perspectives in Computer Science, Palgrave, Houndsmill, Hampshire, 2000, pp. 303–321.
[17] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 2002, pp. 55–74.
[18] J.C. Reynolds, Class notes for "Specification, Verification, and Refinement of Software", CMU, Spring 2003, Chapter 3, pp. 51–61, avaliable from the author's homepage.
[19] H. Schorr, W.M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Commun. ACM 10 (8) (1967) 501–506.
[20] N. Suzuki, Automatic verification of programs with complex data structures, PhD thesis, Stanford University (1976), Garland Publishing (1980).
[21] M. Wenzel, Isabelle/isar – a versatile environment for human-readable formal proof documents, PhD thesis, Institut für Informatik, Technische Universität München. Available from: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html (2002).
[22] H. Yang, Local reasoning for stateful programs, PhD thesis, University of Illinois, Urbana-Champaign (2001).